BLG 562E – Term Project Progress Report High-Performance Parallelization of FMCW Radar Signal Processing on GPU

Bahadır Çeliktaş 504251244

December 19, 2025

1 Introduction

Problem Definition: Frequency Modulated Continuous Wave (FMCW) radar systems require intensive signal processing steps, primarily Fast Fourier Transforms (FFT) and matrix transpositions, to determine the range and velocity of targets [2]. In real-time applications such as autonomous driving or drone navigation, these computations must be performed with extremely low latency. Standard CPU implementations often fail to meet these real-time constraints as the data resolution and radar channel count increase.
Motivation: The primary motivation is to leverage the massive parallelism of modern GPU architectures (specifically NVIDIA Ampere) to accelerate the FMCW processing pipeline. By offloading computationally heavy tasks like 2D-FFT from the CPU to the GPU, we aim to achieve significant speedups. Furthermore, this project investigates the performance gap between naive implementations and hardware-aware optimizations (Shared Memory, AVX SIMD), analyzing where the true bottlenecks lie (Compute vs. Memory Transfer).

2 Related Work

The Fast Fourier Transform (FFT) is the core algorithm for radar signal processing. While the Cooley-Tukey algorithm is the standard for sequential processing [1], parallel implementations vary significantly based on hardware architecture.

State-of-the-art solutions typically rely on vendor-optimized libraries like NVIDIA cuFFT [4] or Intel MKL. However, custom kernel implementations are crucial for understanding hardware limits as detailed in the CUDA Programming Guide [3], and for optimizing specific pipeline stages where generic libraries might introduce overhead. Existing literature often compares CPU vs. GPU but rarely details the step-by-step impact of optimizations like register pressure reduction and shared memory banking on modern RTX series cards.

3 Proposed Method

We propose a step-by-step optimization pipeline to parallelize the Range-Doppler map generation algorithm. The implementation stages are as follows:

- **CPU Baselines:** Implementing Recursive FFT (Reference), OpenMP (Multi-thread), and AVX (SIMD) versions to establish strong baselines.
- **GPU Naive Implementation:** Direct porting of the algorithm to CUDA using Global Memory.
- **GPU Optimized Implementation:**
  - Utilizing **Shared Memory** to reduce global memory latency during FFT stages.
  - Implementing a custom **Bit-Reversal** kernel.

- Optimizing **Matrix Transpose** with memory padding to avoid bank conflicts.
- Precision optimization (Double to Float conversion) to utilize FP32 CUDA cores effectively on RTX 3060.

- **Comparison:** Benchmarking against the highly optimized NVIDIA cuFFT library.

4 Experimental Results

The experiments were conducted on an NVIDIA RTX 3060 GPU and a generic multi-core CPU. We measured "Compute Time" (kernel execution), "Total Time" (including memory transfers), "Effective Bandwidth," and "RMSE" for validation. The CPU AVX implementation was used as the ground truth reference for accuracy measurements.

4.1 Performance Analysis

Table 1 presents the execution times and bandwidth utilization for different implementation stages.

Table 1: Performance and Accuracy Comparison (Validation vs. CPU AVX)

Implementation	**Compute (ms)**	**Total (ms)**	**Bandwidth (GB/s)**	**Speedup (vs CPU Rec.)**	**RMSE Error**
CPU Recursive	683.85	∼684	N/A	1x (Baseline)	-
CPU OpenMP	249.18	249.18	N/A	2.7x	-
CPU AVX (Ref)	9.28	9.28	N/A	74.5x	**Reference**
GPU Global (Naive)	6.42	9.85	2.61	69.4x	N/A
GPU Shared (Custom)	**0.58**	**4.32**	**28.68**	**158.3x**	**0.001048**
GPU 1D FFT + Manual Transpose	0.32	3.87	52.85	176.7x	0.000513
GPU cuFFT (Vendor)	0.25	1.80	67.38	380.0x	0.000528

4.2 Bandwidth & Bottleneck Discussion

The naive GPU implementation suffered from uncoalesced memory access patterns, achieving only **2.61 GB/s**, which is less than 1% of the RTX 3060's theoretical peak bandwidth (360 GB/s). By utilizing Shared Memory to localize data on-chip, we achieved a **11x improvement** in effective bandwidth (**28.68 GB/s**).

It is observed that the **GPU 1D FFT + Manual Transpose** stage achieved significantly higher bandwidth (**52.85 GB/s**) compared to the full 2D FFT kernel. This is attributed to the lower arithmetic intensity of the transpose operation compared to the complex butterfly operations required for the full FFT. While the custom kernel is extremely fast (0.58 ms compute), the total time is dominated by PCIe transfer overhead (∼3.7 ms), confirming that the system is now bandwidth-bound rather than compute-bound.

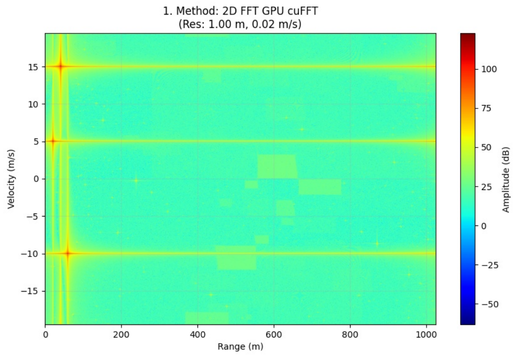
4.3 Validation (RMSE and Visual Analysis)

To ensure the correctness of the parallel algorithms, we calculated the Root Mean Square Error (RMSE) against the CPU AVX output.

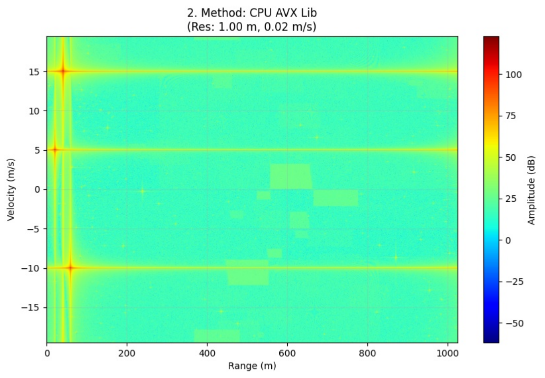
- The **GPU Shared Memory FFT** yielded an RMSE of **0.0010**, which is negligible for radar signal processing and falls within the expected range for single-precision (FP32) floating-point arithmetic differences.

- The **GPU 1D FFT + Manual Transpose** and **cuFFT** implementations showed even lower error rates (∼0.0005), confirming high data integrity during matrix operations.

In addition to numerical verification, visual validation was performed by generating Range-Doppler maps. Figure 1 illustrates the output of the GPU implementation compared to the CPU AVX reference. The identical placement of targets and consistent noise floor visually confirm the correctness of the algorithm.



(a) Method: 2D FFT GPU cuFFT



(b) Method: CPU AVX Lib

Figure 1: Comparison of Range-Doppler Maps generated by GPU (Left) and CPU AVX (Right). The resolution is 1.00 m in Range and 0.02 m/s in Velocity.

5 Conclusion

We have successfully implemented and optimized the FMCW radar algorithms on the GPU. The transition from naive Global Memory usage to Shared Memory resulted in dramatic compute speedups and bandwidth improvements. However, our analysis reveals that as computation becomes extremely fast, the memory transfer over PCIe bus becomes the dominant latency factor.

The following steps are planned to be accomplished before submitting the final report:

- **Step 1: Pinned Memory Optimization.** Implementing `cudaMallocHost` (Pinned/Page-locked memory) to maximize PCIe bandwidth and reduce the transfer overhead observed in the results.
- **Step 2: CUDA Streams.** Overlapping data transfer with kernel execution (pipelining) to hide latency.
- **Step 3: Real Data Validation.** Testing the pipeline with larger, real-world ADC datasets to observe scaling behavior.

References

[1] J. W. Cooley and J. W. Tukey, ”An algorithm for the machine calculation of complex Fourier series,” *Mathematics of Computation*, vol. 19, no. 90, pp. 297–301, 1965.

[2] M. A. Richards, *Fundamentals of Radar Signal Processing*, 2nd ed. New York, NY, USA: McGraw-Hill Education, 2014.

[3] NVIDIA Corporation, ”NVIDIA CUDA C++ Programming Guide,” Version 12.3, 2023. [Online]. Available: `https://docs.nvidia.com/cuda/cuda-c-programming-guide/`

[4] NVIDIA Corporation, ”cuFFT Library User’s Guide,” 2023. [Online]. Available: `https://docs.nvidia.com/cuda/cufft/`